Single inverted pendulum
EV5 - Modelling and Control

Arne van Iterson

February 11, 2024

Abstract

Electrical engineering students at the University of applied sciences Utrecht are assigned to simulate, build and control an inherently unstable system during the 'Modelling and Control' course of the third year. This paper describes the process of doing just that for a single inverted pendulum. The pendulum itself is built using LEGO® Mindstorms EV3. The control system is a PID-loop is implemented using Python. A digital twin, also built in Python, is used to simulate the system.

1 Theory

The system that will be discussed in this paper is a single inverted pendulum. Common implementations of this system include a cart of some sort that carries the pendulum on a pivot point over a defined path.

While the system could theoretically stand up straight and never fall over, in practice the system is inherently unstable and will fall over if not controlled. The system can be controlled using a motor that moves the cart up and down the path to compensate for the pendulums falling movement. This method of control only requires a sensor to measure the angle of the pendulum.

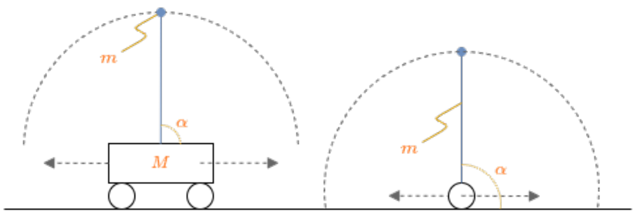


Figure 1: System with (left) and without cart (right)

The system used in this case of this assignment is slightly different; Instead of a cart that carries the pendulum, the pendulum itself is equipped with a motor and the angle is measured using a gyroscope. This difference has been visualised in figure 1. The physics of the system are practically identical, however, it allows for some minor simplifications which will be discussed in section 2.

A practical example of this system is a Segway® People Transporter (figure 2).

Figure 2: Two Segway® PT units in use

2 Model

The physics behind the system in question have been described using the Lagrangian method. The Lagrangian method is a way of describing the dynamics of a system using the kinetic and potential energy of the system.

2.1 Lagrangian

Usually the model for this system is divided into two parts: The cart and the pendulum. The cart is usually considered to be a mass M that moves horizontally along a track. The pendulum is usually considered to be a mass m that is attached to the cart using a pivot point. In this case however, since the motor is attached to the pendulum and the pivot point is the driving shaft of the motor itself, there is no separate mass for the cart. This means that M is equal to m and the formula can be simplified slightly.

The kinetic energy of the system is made up from by the x movement of the system and the falling of the pendulum. Since the system can only move in the x direction using the motor, the kinetic energy is simply:

$$T_1 = \frac{1}{2}mv^2 = \frac{1}{2}m\dot{x}^2 \tag{1}$$

Where x is the displacement of the system in the x direction and m is the mass of the system.

The kinetic energy of the pendulum is influenced by both the x and y movements of the pendulum and therefore:

$$T_2 = \frac{1}{2}m(\dot{x}_p^2 + \dot{y}_p^2)$$

The x and y position of the top of the pendulum can be calculated using the length and angle, this assumes that the pendulum is upright at $\frac{1}{2}\pi$:

$$\begin{aligned} x_p &= x + l \cdot sin(\theta) \\ y_p &= -l \cdot cos(\theta) \end{aligned}$$

The x and y velocity can be derived from the position:

$$\begin{aligned} \dot{x}_p &= \dot{x} + l \cdot cos(\theta) \cdot \dot{\theta} \\ \dot{y}_p &= l \cdot sin(\theta) \cdot \dot{\theta} \end{aligned}$$

Putting this all together gives the kinetic energy of the pendulum:

$$\begin{aligned} T_2 &= \frac{1}{2}m(\dot{x}_s^2 + \dot{y}_s^2) \\ T_2 &= \frac{1}{2}m((\dot{x} + \ell\dot{\theta}cos(\theta))^2 + (\ell\dot{\theta}sin(\theta))^2) \\ T_2 &= \frac{1}{2}m(\dot{x}^2 + 2\dot{x}\ell\dot{\theta}cos(\theta) + \ell^2\dot{\theta}^2cos(\theta)^2 + \ell^2\dot{\theta}^2sin(\theta)^2) \\ T_2 &= \frac{1}{2}m(\dot{x}^2 + 2\dot{x}\ell\dot{\theta}cos(\theta) + \ell^2\dot{\theta}^2(cos(\theta)^2 + sin(\theta)^2)) \end{aligned}$$

Given that $cos(x)^2 + sin(x)^2 = 1$:

$$T_2 = \frac{1}{2}m(\dot{x}^2 + 2\dot{x}\ell\dot{\theta}cos(\theta) + \ell^2\dot{\theta}^2)$$

Combining T_1 and T_2 gives the total kinetic energy of the system:

$$\begin{aligned} T &= T_1 + T_2 \\ T &= \frac{1}{2}m\dot{x}^2 + \frac{1}{2}m(\dot{x}^2 + 2\dot{x}\ell\dot{\theta}cos(\theta) + \ell^2\dot{\theta}^2) \\ T &= m\dot{x}^2 + \frac{1}{2}m(2\dot{x}\ell\dot{\theta}cos(\theta) + \ell^2\dot{\theta}^2) \\ T &= m\dot{x}^2 + m\dot{x}\ell\dot{\theta}cos(\theta) + \frac{1}{2}m\ell^2\dot{\theta}^2 \end{aligned}$$

The potential energy of the system is:

$$V = -mg\ell cos(\theta) \tag{2}$$

This makes the full Lagrangian of the system:

$$\begin{aligned} \mathcal{L} &= T - V \\ \mathcal{L} &= m\dot{x}^2 + m\dot{x}\ell\dot{\theta}cos(\theta) + \frac{1}{2}m\ell^2\dot{\theta}^2 + mg\ell cos(\theta) \end{aligned} \tag{3}$$

Since the falling pendulum influences the moving pivot point ("cart") and vice versa, the Lagrangian will have to be solved for both the angular acceleration $\ddot{\theta}$ of the pendulum and the horizontal acceleration of the pivot $\ddot{x}$

2.2 Horizontal acceleration

The horizontal acceleration of the pivot point can be derived from the Lagrangian using the following formula:

$$\frac{d}{dt}\left(\frac{\partial\mathcal{L}}{\partial\dot{x}}\right) = \frac{\partial\mathcal{L}}{\partial x} \tag{4}$$

Filling in for $\dot{x}$

$$\begin{aligned} & m\dot{x}^2 + m\dot{x}\ell\dot{\theta}cos(\theta) \\ \frac{\partial\mathcal{L}}{\partial\dot{x}} &= 2m\dot{x} + m\ell\dot{\theta}cos(\theta) \\ \frac{d}{dt}\left(\frac{\partial\mathcal{L}}{\partial\dot{x}}\right) &= 2m\ddot{x} + m\ell\ddot{\theta}cos(\theta) - m\ell\dot{\theta}sin(\theta) \end{aligned} \tag{5}$$

Filling in for x

$$\frac{\partial\mathcal{L}}{\partial x} = 0 \tag{6}$$

Combining formulas 5 and 6 using formula 4 gives:

$$2m\ddot{x} + m\ell\ddot{\theta}cos(\theta) - m\ell\dot{\theta}sin(\theta) = 0 \tag{7}$$

Which gives:

$$\begin{aligned} 2m\ddot{x} &= -m\ell\ddot{\theta}cos(\theta) + m\ell\dot{\theta}sin(\theta) \\ \ddot{x} &= -\frac{m\ell\ddot{\theta}cos(\theta) + m\ell\dot{\theta}sin(\theta)}{2m} \end{aligned} \tag{8}$$

2.3 Angular acceleration

$$\frac{d}{dt}\left(\frac{\partial\mathcal{L}}{\partial\dot{\theta}}\right) = \frac{\partial\mathcal{L}}{\partial\theta} \tag{9}$$

Filling in for $\dot{\theta}$

$$\begin{aligned} & m\dot{x}\ell\dot{\theta}cos(\theta) + \frac{1}{2}m\ell^2\dot{\theta}^2 \\ \frac{\partial\mathcal{L}}{\partial\dot{\theta}} &= m\dot{x}\ell cos(\theta) + m\ell^2\dot{\theta} \\ \frac{d}{dt}\left(\frac{\partial\mathcal{L}}{\partial\dot{\theta}}\right) &= m\ddot{x}\ell cos(\theta) - m\dot{x}\ell sin(\theta) + m\ell^2\ddot{\theta} \end{aligned} \tag{10}$$

Filling in for θ

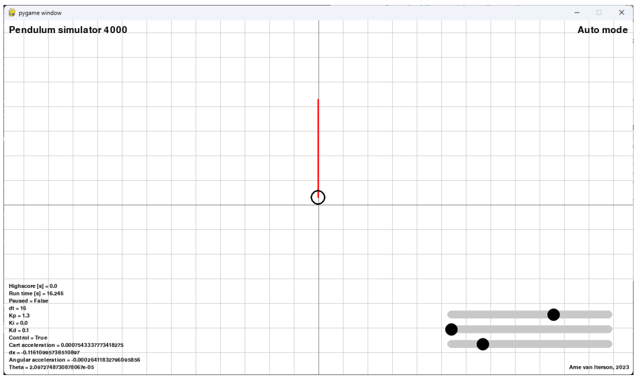
$$\begin{aligned} & m\dot{x}\ell\dot{\theta}cos(\theta) + mg\ell cos(\theta) \\ \frac{\partial\mathcal{L}}{\partial\theta} &= -m\dot{x}\ell\dot{\theta}sin(\theta) - mg\ell sin(\theta) \end{aligned} \tag{11}$$

Combining formulas 10 and 11 using formula 9 gives:

$$\begin{aligned} m\ell^2\ddot{\theta} &= -m\ddot{x}\ell cos(\theta) + m\dot{x}\ell sin(\theta) - m\dot{x}\ell\dot{\theta}sin(\theta) - mg\ell sin(\theta) \\ \ell\ddot{\theta} &= -\ddot{x}cos(\theta) + \dot{x}sin(\theta) - \dot{x}\dot{\theta}sin(\theta) - gsin(\theta) \\ \ddot{\theta} &= -\frac{\ddot{x}cos(\theta) - \dot{x}sin(\theta) + \dot{x}\dot{\theta}sin(\theta) + gsin}{l} \end{aligned} \tag{12}$$

2.4 Simulation

The formula's in the previous sections have then been put into a purpose built Python simulation suite called Pendulum Simulator 4000. The source code for this can be found in the appendix and in a Git repository, which can be found at `https://arnweb.nl/gitea/arne/EV5_Modcon`.



```
Highscore [s] = 0.0
Run time [s] = 16.246
Paused = False
dt = 16
Kp = 1.3
Ki = 0.0
Kd = 0.1
Control = True
Cart acceleration = 0.0007543337773418275
dx = -0.11610995738510897
Angular acceleration = -0.00026411832796095856
Theta = 2.0972748730878067e-05
```

Figure 3: Main window of Pendulum Simulator 4000

3 Setup

To simplify hardware design, the system will be built and run on a LEGO® Mindstorms EV3 development kit. The programmable brick itself, hereinafter referred to as "EV3", runs a Linux distribution known as ev3dev. Ev3dev allows for various programming languages to be used including Python and C. Both of which will be further explained in section 4.

To reproduce the robot in figure 4, please refer to the building instructions. [1].

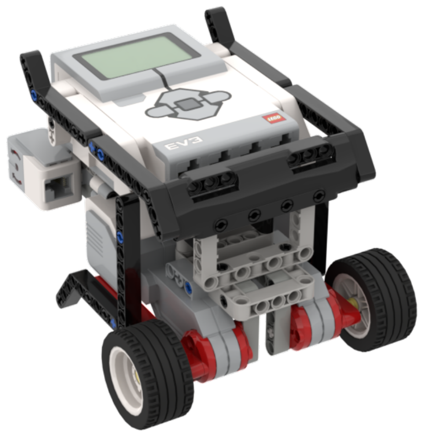


Figure 4: Balancing robot

4 Control

In the simulation it was found that the system was best controlled using a PD-loop, this was to be expected since the system includes integration by it's very nature. The control loop was implemented using Python and the ev3dev library. The source code for this can be found in the appendix.

The choice of hardware has significantly influenced the control system. Unfortunately, it has brought along a lot of issues with the physical model. The gyroscope used has a very limited resolution of 1 degree (1 deg) for angle and 1 degree per second (1 deg/sec) for angular velocity. Add the fact that the sensor drifts about 0,5 deg/sec and the system becomes very unstable very quickly.

This issue would probably have been overcome if the system was easier to program, in its current state the EV3 runs a limited version of Python that takes about two minutes to load every time the program is run. Adding any reasonable logging, through the file system or through dumping python arrays, makes the timing too slow to keep the control loop running fast enough causing the robot to lose balance fairly quickly.

There has been some success in using the C programming language and a compiling toolchain has been set-up, however, the control of simple features like the motor is very low level. The plan was to first get it working in Python and then port it to C but time has unfortunately run out.

In order to get the system to balance at all, the control loop had to be altered to include the position and speed of the motors as has been done in the HTWay project by D. Lechner [2]. The theory behind this is that if the system is still falling in a certain direction while the motors are already moving to compensate that, the movement speed should be increased. In a way the system now uses a cascaded control loop to balance.

A small set of data points has been collected by pre-assigning memory to a numpy array and writing to it directly, then dumping this to non-volatile memory. This gives some insight into the issues with the system, however, this also limits the run time of the system.

5 Validation

The behaviour of the system is as follows:

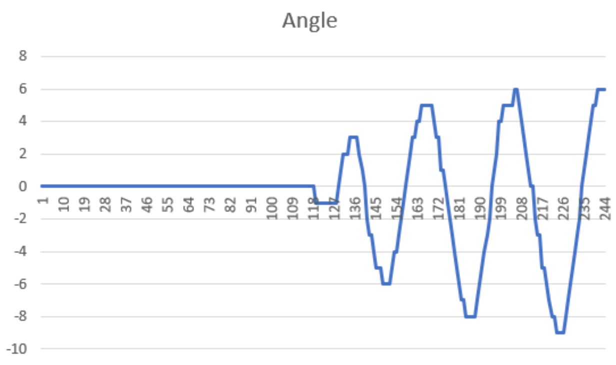


Figure 5: Run 1

The fact that the system oscillates is a typical sign of an ill-tuned control loop. The PID values have been tuned to the best of my ability but the system is still very unstable. Tuning the system has proven difficult due to the inclusion of the motor speed and position in the control loop.

The system is able to say upright using the HTWay project by D. Lechner [2]. Some of the control logic has been used in this project, however the HTWay project includes some filtering logic for the gyro sensor that I did not yet understood well enough to implement.

The fact that I was unable to get the system to balance properly should not change the fact that the simulation should behave the same given the same characteristics and control constants as the physical model, which is not the case.

Running the simulation tuned exactly the same as the physical model reveals an interesting issue; The simulated system is impossible to topple over, which is clearly not the case with the physical model.

The simulation and the model differ in a few ways:

- The simulation does not include sensor drift or resolution
- The simulation does not include the physical limitations of the motor
- The simulation loop runs significantly faster than the physical model does

Drawing any comparisons between the two will be largely useless at this point, so instead we will limit the simulation to check if the simulation will then mimic the actual behaviour.

5.1 Limiting the simulated system

First, the resolution of the gyroscope will be limited to 1 degree. Converting this to radians gives a resolution of 0.0175 rad. The result can be seen in figure 6. Note that the graph has been cropped, showing only a stable part of the simulation.

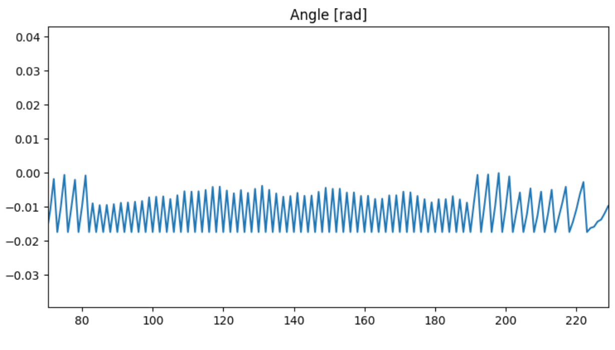


Figure 6: Simulation run with angle resolution limit

The limited resolution does cause the system to oscillate slightly, but not escalating as expected. Additionally, the system is still impossible to topple over.

Next, the simulation will be limited in update speed. This means that the physics simulation will continue while the control loop can only update every certain interval. The log file from the physical model reveals that the average update time for the control loop is about 21 ms or 47 frames per second (fps).

Currently, the simulation runs at 60 fps, the update call for the pendulum control is called every frame. The pendulum control has been changed to only actually control the system after the dt value is more or equal to 21 ms. Because the program runs at 60 fps or roughly 16 ms per frame, the actual PID loop will be called every 32 ms; This is longer than required, but should demonstrate the effect of the delay. The result can be seen in figure 7.

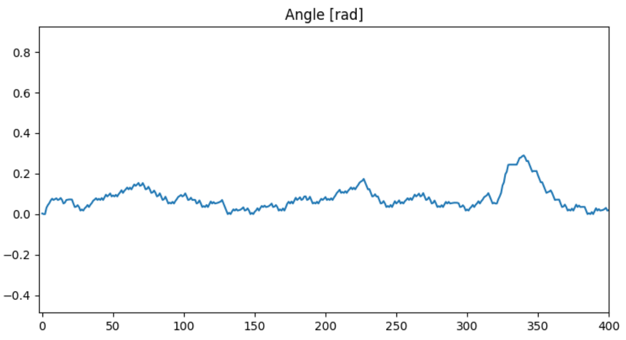


Figure 7: Simulation run with PID control delay

Interestingly, the limited update frequency of the PID controller seems to act as a low-pass filter, reducing the oscillations in the system. The user is now able to topple the system over.

Next, the simulation will be limited in the speed at which it can change the horizontal position of the pendulum. The motor speed on the EV3 is measured in an arbitrary value between -1050 and 1050. The max speed of the motor is about 175 rpm as stated in the research by P. Hurbain. [3]. Combined with the 44 mm diameter of the used wheels gives the maximum speed of the robot as:

$$\begin{aligned} r_{wheel} &= 22[\text{mm}] \\ s_{circumference} &= 2\pi r_{wheel} = 138.23\text{mm} = 0.1382\text{m} \\ v_{max} &= (175/60) \cdot 0.1382 \\ &= 2.92 \cdot 0.1382 = 0.403\text{m/s} \end{aligned} \tag{13}$$

The simulation has been changed to never exceed the speed limit of 0.403 m/s even if the controller asks for it, just like the EV3 motors would do. Limiting the simulation to 0.403 m/s did little to change its behaviour. As can be seen in figure 8, the system never reaches this speed in the first place.

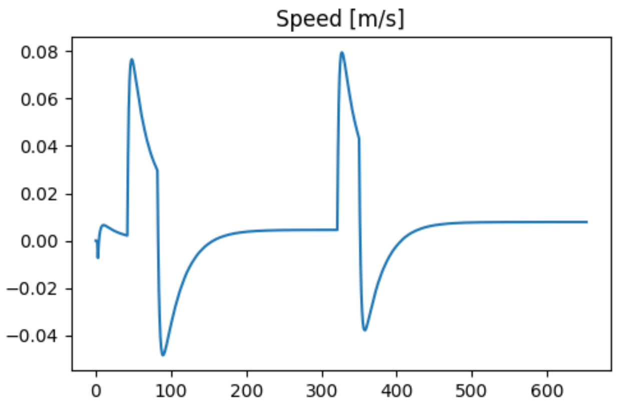


Figure 8: Simulation run with speed limit

Interesting, however, is the acceleration as can be seen in figure 9.

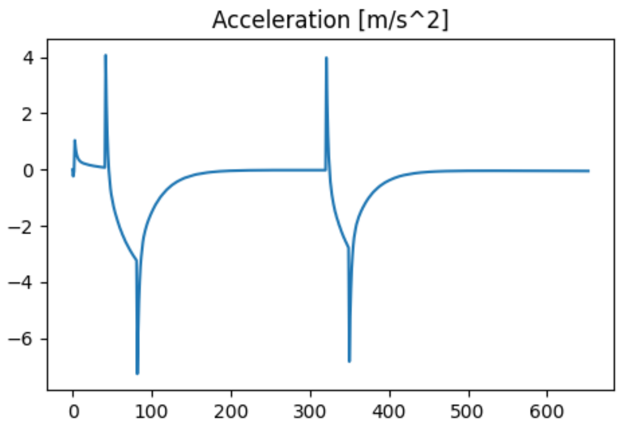


Figure 9: Simulation run with speed limit

The initial kick in acceleration is the attempt to topple the system over by the user. The response is quite harsh, reaching up to 7 m/s^2 which is only slightly impossible for the physical system to achieve. Unfortunately, this is also quite difficult to limit, since the acceleration of the motor would require a measuring setup to determine.

If we make a generous assumption that the motor can reach it's top speed within 0.15 seconds, the acceleration would be:

$$\begin{aligned} v_{max} &= 0.403\text{m/s} \\ t_{max} &= 0.1\text{s} \\ a_{max} &= \frac{v_{max}}{t_{max}} = \frac{0.403}{0.15} = 2.69\text{m/s}^2 \end{aligned} \tag{14}$$

This does cause the system to fall over in the simulation as expected, though not as quickly as the physical model and with less oscillations.

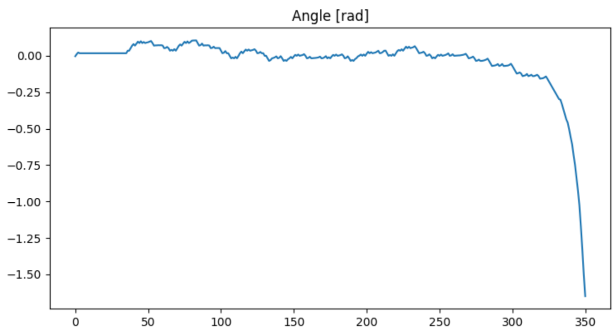


Figure 10: Simulation run with all limits in place

Lastly, the sensor will be offset with about 0,5 degrees per second which will be updated every time the control loop is run. The results can be seen in figure 11.

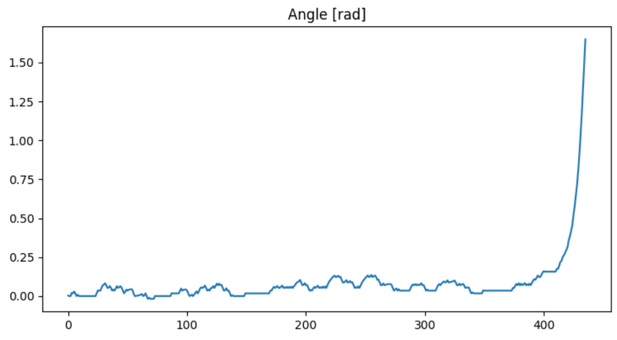


Figure 11: Simulation run with sensor drift

With that, we have successfully created a digital twin for a poorly tuned system. The system is now no longer able to balance properly, it oscillates and falls over just like the physical model.

6 Conclusion

Due to issues with the physical model, this entire project has run slightly backwards. Having to alter the control loop significantly nearing the end of the project has caused the physical model to behave differently than the simulation to a point where comparing them was not really possible. The simulation has been altered to include the limitations of the physical model as closely as possible, which has been successful and the resulting behaviour is comparable to the physical model. So while the system does not balance, at least it makes sense.

7 Recommendations

1. Do not make a robot using LEGO.

8 Ethics of Artificial Intelligence

In Utilitarian ethics the greatest good for the greatest number is the most important. Whatever happens along the way does not matter as long as the outcome is good for the most people or society as a whole. The last couple of years have seen great development in the field of Artificial Intelligence, which brings along the question of how and if this technology should be used from a utilitarian perspective.

The answer to this question is not as simple as it may seem. On the one hand, AI can be used to automate tasks that are dangerous or tedious for humans. This would allow them to focus on more important tasks and would reduce the amount of work related injuries; Which would be favourable according to utilitarian ethics. This also means however, that it would replace work that is currently done by humans, therefore leading to a loss of jobs and, potentially, a group of people that are unable to find work.

Then again, we are currently experiencing the development of AI, which means that what is happening now might just be part of the AI development that will eventually lead to the greatest thing humans have ever done that will eventually benefit everyone. In that case, we might just have to accept the loss of jobs and the other negative effects of AI development as a necessary evil until the great-for-everyone endpoint is reached.

This great-for-everyone endpoint however, is not a certainty and if it is ever reached, it might take a very long time. Therefore, I am of the opinion that not all types of AI should be treated with the same ethical standards.

In the context of modelling and control, I think most of the work described in this paper, maybe with the exception of building the physical model itself, will no longer be done by a person in the near future. Writing this paper and parts of the software in VSCode with Github Copilot Enabled has taught me that with a more limited context compared to generic AI, like ChatGPT, the AI 'understood' more complex code structures and physics than I expected. At the moment, I don't think it would be a great idea to run Copilot code without any checking by a human, but I do think that it will be possible in the near future.

There is a lot more to engineering than just coding, therefore I don't think engineers as a whole will be replaced at all. AI development towards the aid of developers will likely not directly cost any jobs. There have been cases where it has or likely will, especially creative jobs, journalism and administrative jobs. [4] This type of development, I believe, should not stop but should be kept in check. We have seen the writers strikes and the companies firing their entire support staff after OpenAI opened the GPT-3 API. [5] While it might be part of the upcoming great-for-everyone endpoint, right now I do not believe we should allow AI to completely replace jobs with half-baked solutions for the sake of profit.

References

[1] A. van Iterson, “Building instructions.” [Online]. Available: https://arnweb.nl/gitea/arne/EV5_Modcon/releases/download/pre-alpha/Instruction.pdf

[2] D. Lechner, “Python port of the hitechnic htway for ev3dev,” Github, 2010. [Online]. Available: https://gist.github.com/dlech/11098915

[3] P. Hurbain, “Lego® 9v technic motors compared characteristics,” 2012. [Online]. Available: https://www.philohome.com/motors/motorcomp.htm

[4] L. A. Times, “Writers’ strike: What happened, how it ended and its impact on hollywood.” [Online]. Available: https://www.latimes.com/entertainment-arts/business/story/2023-05-01/writers-strike-what-to-know-wga-guild-hollywood-productions

[5] W. Post, “Chatgpt provided better customer service than his staff. he fired them.” [Online]. Available: https://www.washingtonpost.com/technology/2023/10/03/ai-customer-service-jobs/

A Simulation source code

A.1 sim.py

```
# Pendulum simulator 4000
# Arne van Iterson, 2023

# Imports
import pygame_widgets
import pygame
from pygame_widgets.slider import Slider
from pygame.math import Vector2
import math

# pygame setup
pygame.init()
screen = pygame.display.set_mode((1280, 720))
clock = pygame.time.Clock()
running = True
update = True
pole = Vector2(screen.get_rect().center)  # center of screen

# Own objects must be imported after pygame init
from pendulum import Pendulum
from uiHelpers import *

# UI helpers
ui = SimUI(screen, pole)

# Pendulum setup
# Start angle in radians, length, mass, color
pendulum = Pendulum(0, 0.2, 0, 0.7, ”red”)
pendulum.reset()
dx = 0  # x offset
dt = 1  # delta time

# Sliders
slider_kp = Slider(screen, 910, 590, 320, 16, initial=pendulum.kp, min=0, max=
slider_ki = Slider(screen, 910, 620, 320, 16, initial=pendulum.ki, min=0, max=
slider_kd = Slider(screen, 910, 650, 320, 16, initial=pendulum.kd, min=0, max=

# Gametime
rt = 10  # run time
highscore = 0


# Metadata values
def meta():
    ui.meta(pendulum.theta[pendulum.index], ”Theta”)
    ui.meta(pendulum.a_ang[pendulum.index], ”Angular␣acceleration”)
    ui.meta(pendulum.dx, ”dx”)
    ui.meta(pendulum.a_cart[pendulum.index], ”Cart␣acceleration”)

    ui.meta(pendulum.pid, ”Control”)

    ui.meta(pendulum.kd, ”Kd”)
    ui.meta(pendulum.ki, ”Ki”)
    ui.meta(pendulum.kp, ”Kp”)

    ui.meta(dt, ”dt”)
    ui.meta(not update, "Paused")
    ui.meta(rt / 1000, "Run time [s]")
    ui.meta(highscore / 1000, "Highscore [s]")


while running:
    events = pygame.event.get()
    ### User controls ###
    for event in events:
        # Quit
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            # Quit
            if event.key == pygame.K_ESCAPE:
                running = False
            # Reset simulation
            elif event.key == pygame.K_SPACE:
                pendulum.reset()
                rt = 0
            # Pause simulation
            elif event.key == pygame.K_p:
                update = not update
            # Display plot if simulation is not running
            elif event.key == pygame.K_g:
                if pendulum.fallen:
                    pendulum.plot()
                else:
                    update = False
                    pendulum.plot()
            # Toggle PID controller
            elif event.key == pygame.K_c:
                pendulum.pid = not pendulum.pid

    # Move pendulum
    keys = pygame.key.get_pressed()
    if keys[pygame.K_LEFT] or keys[pygame.K_a]:
        pendulum.a_cart[pendulum.index] -= 4
    if keys[pygame.K_RIGHT] or keys[pygame.K_d]:
        pendulum.a_cart[pendulum.index] += 4

    # Draw grid
    ui.grid(50, 0, 15)

    # Update PID values
    # pendulum.kp = slider_kp.getValue()
    # pendulum.ki = slider_ki.getValue()
    # pendulum.kd = slider_kd.getValue()

    # Update pendulum
    if not pendulum.fallen:
        if update:
            rt += dt
            pendulum.update(dt)
    else:
        ui.gameover(rt)

        # Update highscore
        if rt > highscore:
            highscore = rt
        # Draw metadata
        ui.update(dt)
        meta()

        # Draw pendulum
        dx = (pendulum.dx, 0)
        pygame.draw.line(screen, pendulum.color, pole + dx, pole + pendulum.vector
        pygame.draw.circle(screen, "black", pole + dx, 15, 3)

        # Draw frame
        pygame_widgets.update(events)
        pygame.display.flip()
        dt = clock.tick(60)  # limits FPS to 120

pygame.quit()
```

A.2 pendulum.py

```
from pygame.math import Vector2
import math
import numpy as np
import random
import matplotlib.pyplot as plt

# Constants
C_GRAVITY = 9.81  # m/s^2
C_MTPRATIO = 100  # Pixels per meter
C_P_ANG_START = 1 / 1000 * math.pi
C_FALL_ANG = 52.5 / 100 * math.pi

C_ANG_RES_LIMIT = ((2 * math.pi) / 360) # rad
C_V_CART_LIMIT = 0.403  # m/s
C_A_CART_LIMIT = C_V_CART_LIMIT / 0.1  # m/s^2

class Pendulum:
    def __init__(self, theta, length, dx, mass, color):
        """
        Initialize a Pendulum object.

        Parameters:
        theta (float): Angle [rad].
        length (float): Length of the pendulum [m].
        dx (float): Horizontal displacement of the "cart" from the center [m].
        mass (float): Mass of the pendulum for physics calculations [kg].
        color (str): Display color.

        Returns:
        None
        """
        ## Game variables
        self.vector = None  # Vector2 object
        self.fallen = False # Stop when pendulum falls over

        ## Physics variables
        self.index = 0 # Index helper for plotting graphs
        self.theta = [theta]  # Angle in radians
        self.a_ang = [0]  # Angular acceleration
        self.v_ang = [0]  # Angular velocity

        ## Gyro offset
        self.theta_offset = 0

        self.dx = dx  # Horizontal displacement of "cart" from center
        self.a_cart = [0]  # Acceleration of cart
        self.v_cart = [0]  # Velocity of cart
        self.s_cart = [0]  # Displacement of cart [m]

        # self.r_factor = 0.50  # Damping factor

        self.length = length  # Length of pendulum
        self.mass = mass  # Mass of pendulum for physics
        self.color = color  # Display color

        ## PID variables
        self.pid = True
        self.pidDelay = 0
        # self.kp = 1.3
        # self.ki = 0.0
        # self.kd = 0.1
        self.kp = 7.5
        self.ki = 0.0
        self.kd = 1.15

        self.kp_m = 0.07
        self.kd_m = 0.1

    def update(self, dt):
        self.doMath(dt)
        self.vector = Vector2.from_polar(
            ((self.length * C_MTPRATIO), math.degrees(self.theta[self.index] -
        )

        if self.pid:
            self.pidDelay += dt

            if self.pidDelay > 21:
                self.pidControl(self.pidDelay)
                self.theta_offset += self.pidDelay * (C_ANG_RES_LIMIT / 1000)
                self.pidDelay = 0

        if abs(self.theta[self.index]) == C_FALL_ANG:
            self.fallen = True

    # def update(self, dt):
    #     """
    #     Update the pendulum's state based on the elapsed time.

    #     Parameters:
    #     - dt (float): The elapsed time in milliseconds.

    #     Returns:
    #     None
    #     """
    #     a_ang = (-(C_GRAVITY * math.sin(self.theta)) / (self.length)) - (sel
# Angular acceleration

    #     v_ang = a_ang * (dt/1000) + self.v_ang  # Integrate acceleration to
    #     s_ang = v_ang * (dt/1000)  # Angular displacement

    #     self.theta += s_ang  # Update value
    #       self.vector = Vector2.from_polar(((self.length * 150), math.degrees(

    #       self.a_ang = a_ang  # Update value
    #       self.v_ang = v_ang  # Update value

    def doMath(self, dt):
        ### ANGLE ###
        ang_term1 = self.a_cart[self.index] * math.cos(self.theta[self.index] +
        ang_term2 = self.v_cart[self.index] * math.sin(self.theta[self.index] +
        ang_term3 = (
            self.v_cart[self.index]  # Previous cart velocity
            * self.v_ang[self.index]  # previous angle velocity
            * math.sin(self.theta[self.index] + self.theta_offset)  # Sin prev
        )
        ang_term4 = C_GRAVITY * math.sin(self.theta[self.index] + self.theta_o

        # Angular acceleration
        self.a_ang.append(
            (ang_term1 - ang_term2 + ang_term3 - ang_term4) / -(self.length)
        )

        # Integrate acceleration to get velocity
        self.v_ang.append(
            self.v_ang[self.index]  # Previous velocity
            + (self.a_ang[self.index + 1] * (dt / 1000))
        )

        # Angular displacement
        theta_res = self.theta[self.index] + (self.v_ang[self.index + 1] * (dt
        theta_round = theta_res % C_ANG_RES_LIMIT
        if theta_round > 0.5 * C_ANG_RES_LIMIT:
            theta_res = theta_res + theta_round
        else:
            theta_res = theta_res - theta_round
        self.theta.append(theta_res)

        # Limit fall of pendulum
        self.theta[self.index + 1] = self.clamp(
            self.theta[self.index + 1], -C_FALL_ANG, C_FALL_ANG
        )

        ### CART ###
        cart_term1 = (
            self.mass  # Mass
            * self.length  # Length
            * self.a_ang[self.index + 1]  # Current angle acceleration
            * math.cos(self.theta[self.index + 1])  # Current angle
        )
        cart_term2 = (
            self.mass  # Mass
            * self.length  # Length
            * self.v_ang[self.index + 1]  # Current angle velocity
            * math.sin(self.theta[self.index + 1])  # Current angle
        )

        # Cart acceleration
        a_res = (-cart_term1 + cart_term2) / (2 * self.mass)
        self.a_cart.append(self.clamp(a_res, -C_A_CART_LIMIT, C_A_CART_LIMIT))
        # if abs(a_res) < C_A_CART_LIMIT:
        #       self.a_cart.append(a_res)
        # else:
        #     self.a_cart.append(C_A_CART_LIMIT * (a_res / abs(a_res)))

        # Integrate acceleration to get velocity
        v_res = self.v_cart[self.index] + (self.a_cart[self.index + 1] * (dt /

        if abs(v_res) < C_V_CART_LIMIT:
            self.v_cart.append(v_res)
        else:
            self.v_cart.append(C_V_CART_LIMIT * (v_res / abs(v_res)))

        # Cart displacement
        self.s_cart.append(
            self.s_cart[self.index]  # Previous displacement
            + (self.v_cart[self.index + 1] * (dt / 1000))
        )
        self.dx = self.s_cart[self.index + 1] * C_MTPRATIO  # Convert to pixel

        # Update index
        self.index += 1

    def clamp(self, n, minn, maxn):
        return max(min(maxn, n), minn)

    def reset(self):
        self.index = 0

        self.a_ang = [0]
        self.v_ang = [0]
        self.dx = [0]
        self.theta = [random.choice([1, -1]) * C_P_ANG_START]

        self.a_cart = [0]
        self.v_cart = [0]
        self.s_cart = [0]
        self.theta_offset = 0
        self.fallen = False
        self.update(0)

    def plot(self):
        fig, axs = plt.subplots(2, 2)
        fig.suptitle("Pendulum")

        axs[0,0].plot(self.theta)
        axs[0,0].set_title('Angle [rad]')
        axs[0,1].plot(self.v_ang)
        axs[0,1].set_title('Angular velocity [rad/s]')
        axs[1,0].plot(self.a_ang)
        axs[1,0].set_title('Angular acceleration [rad/s^2]')

        fig, axs = plt.subplots(2, 2)
        fig.suptitle("Cart")

        axs[0,0].plot(self.s_cart)
        axs[0,0].set_title('Position [m]')
        axs[0,1].plot(self.v_cart)
        axs[0,1].set_title('Speed [m/s]')
        axs[1,0].plot(self.a_cart)
        axs[1,0].set_title('Acceleration [m/s^2]')

        plt.show()
    def pidControl(self, dt):
        error = self.theta[self.index]
        dt = (dt/1000)
        # result = (self.kp * error) + (self.ki * sum(self.theta)) + (self.kd
        result = (self.kp * error) \
                + (((error - self.theta[self.index - 1]) / (dt + 1 / 1000)) \
                * self.kd) \
                + (self.kp_m * self.s_cart[self.index]) \
                + (self.kd_m * self.v_cart[self.index])
        self.a_cart[self.index] = self.clamp(result * 10, -C_A_CART_LIMIT, C_A
```

A.3 uiHelpers.py

```
import pygame

# Constants
C_GRID_L_VALUE = 200
C_GRID_D_VALUE = 100
C_MPLOT_START = 700
C_BLINK_TIME = 500

gridLight = pygame.Color(C_GRID_L_VALUE, C_GRID_L_VALUE, C_GRID_L_VALUE)
gridDark = pygame.Color(C_GRID_D_VALUE, C_GRID_D_VALUE, C_GRID_D_VALUE)
font_h = pygame.font.SysFont(None, 28)
font_m = pygame.font.SysFont(None, 16)

# UI Class
class SimUI:
    def __init__(self, screen, pole):
        self.screen = screen
        self.pole = pole

        self.metaPlotY = C_MPLOT_START

        self.blink = False
        self.blinkTimer = 0

        self.controlled = True
        self.paused = False

    def meta(self, val, desc):
        # Capture values to display status
        if desc == "Control":
            self.controlled = val
        if desc == "Paused":
            self.paused = val

        # Print em
        self.screen.blit(
            font_m.render(f"{desc} = {val}", True, "black"), (10, self.metaPlo
        )
        self.metaPlotY -= 15

    def grid(self, dist, Xoff=0, Yoff=0):
        # Clear the screen
        self.screen.fill("white")

        # Drawing offsets so the grid aligns with the pole
        cXoff = self.pole.x % dist
        cYoff = self.pole.y % dist
        # Draw the grid
        for i in range(0, 1280, dist):
            pygame.draw.line(
                self.screen,
                gridLight,
                (i + Xoff + cXoff, 0),
                (i + Xoff + cXoff, 720),
                1,
            )
            pygame.draw.line(
                self.screen,
                gridLight,
                (0, i + Yoff + cYoff),
                (1280, i + Yoff + cYoff),
                1,
            )

        # Draw the center lines darker
        pygame.draw.line(
            self.screen, gridDark, (self.pole.x + Xoff, 0), (self.pole.x + Xof
        )
        pygame.draw.line(
            self.screen,
            gridDark,
            (0, self.pole.y + Yoff),
            (1280, self.pole.y + Yoff),
            1,
        )

    def centeredText(self, font, text="", colour="black", y=0):
        textObj = font.render(text, True, colour)
        text_rect = textObj.get_rect(center=(1280 / 2, 720 / 2 - y))
        self.screen.blit(textObj, text_rect)

    def gameover(self, time):
        font_g = pygame.font.Font("res\\pricedown.otf", 128)
        self.centeredText(font_g, "wasted", "red", 150)
        self.centeredText(font_m, f"You␣controlled␣the␣pendulum␣for␣{time␣/␣10

        self.centeredText(font_m, "Press␣space␣to␣restart", "black", 60)
        self.centeredText(font_m, "Press␣G␣to␣view␣nerd␣graphs", "black", 45)

    def update(self, dt):
        # Credits
        self.screen.blit(
            font_h.render("Pendulum␣simulator␣4000", True, "black"), (10, 10)
        )
        self.screen.blit(
            font_m.render("Arne␣van␣Iterson,␣2023", True, "black"), (1150, 700
        )

        # Reset meta printing
        self.metaPlotY = C_MPLOT_START

        # Draw current status blinking
        if self.blink:
            if self.paused:
                text = "Paused"
            elif self.controlled:
                text = "Auto␣mode"
            else:
                text = "Manual mode"

            textObj = font_h.render(text, True, "black")
            text_rect = textObj.get_rect(right=1270, top=10)
            self.screen.blit(textObj, text_rect)

        if self.blinkTimer > C_BLINK_TIME:
            self.blink = not self.blink
            self.blinkTimer = 0
        self.blinkTimer += dt
```

B Control source code

```
#!/usr/bin/env python3

# https://github.com/ev3dev/ev3dev-lang-python
# https://github.com/ev3dev/ev3dev-lang-python-demo#balanc3r

from ev3dev2.motor import LargeMotor, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor import INPUT_2
from ev3dev2.sensor.lego import GyroSensor
from ev3dev2.sound import Sound

import time
# import logging
import os
import csv

from datetime import datetime

now = datetime.now() # current date and time
# LOGNAME = os.path.join(os.getcwd(), "logs/", (now.strftime("%m-%d_%H:%M:%S") + "
# # print(LOGNAME)
# logging.basicConfig(filename=LOGNAME,
#                     filemode='a',
#                     format='%(message)s',
#                     level=logging.DEBUG)

log_t = []
log_ang = []
log_vang = []
log_x = []
log_v = []

# Motors
l_motor = LargeMotor(OUTPUT_B)
r_motor = LargeMotor(OUTPUT_C)

# Sensors
gyro = GyroSensor(INPUT_2)

# Sound
sound = Sound()

# tires are 56mm diameter
def balance(target_angle=0):
    t = time.time()

    # PID
    gyro_k_p = 7.5
    # float k_i = 0 # No integral in this system
    gyro_k_d = 1.15
    motor_k_p = 0.07
    motor_k_d = 0.1

    # Calibrate gyro in current position
    gyro.calibrate()
    gyro.mode = GyroSensor.MODE_GYRO_G_A
    angle = gyro.angle

    # Calibrate motor
    prev_sum_motor_pos = 0
    l_motor.reset()
    r_motor.reset()
    pos = 0

    sound.speak('3, 2, 1')

    # Stop if the robot has fallen over
    while abs(angle) < 40:
        # Keep time
        dt = time.time() - t
        t = time.time()

        sum_motor_pos = l_motor.position + r_motor.position

        delta_motor_pos = sum_motor_pos - prev_sum_motor_pos
        pos += delta_motor_pos
        speed = (delta_motor_pos / dt)

        prev_sum_motor_pos = sum_motor_pos

        angle, rate = gyro.angle_and_rate

        log_t.append(t)
        log_ang.append(angle)
        log_vang.append(rate)
        log_x.append(pos)
        log_v.append(speed)

        error = angle - target_angle
        pd = gyro_k_p * error + gyro_k_d * rate \
            + motor_k_p * pos + motor_k_d * speed

        # convert -100 ~ 100 to -1050 ~ 1050
        speed = (((pd - (-100)) * (1049 - (-1049))) / (100 - (-100))) + (-1049)
        l_motor.run_forever(speed_sp=speed)
        r_motor.run_forever(speed_sp=speed)

        # logging.debug("%d;%d;%d", angle, rate, pd)

        # if abs(pd) > 1050:
        #     print("cap")

        # speed = min(max(pd, -1050), 1050)

    l_motor.stop(stop_action="hold")
    r_motor.stop(stop_action="hold")


# Combine the arrays
try:
    balance()
except:
    l_motor.stop(stop_action="hold")
    r_motor.stop(stop_action="hold")
finally:
    data = zip(log_t, log_ang, log_vang, log_x, log_v)

    # Define the filename
    now = datetime.now() # current date and time
    LOGNAME = os.path.join(now.strftime("%m-%d_%H:%M:%S") + "new.csv")
```

```
            # Write data to CSV file
            with open(LOGNAME, 'w', newline='') as csvfile:
                csvwriter = csv.writer(csvfile)
                csvwriter.writerow(['Time', 'Angle', 'Angular-Velocity', 'X-pos', 'Speed'])
# Write header row
                csvwriter.writerows(data)
```